INSIDE VISUAL dBASE™

September 1996 • Vol. 3 No. 9
US $8.50

Tips & techniques for Visual dBASE and dBASE for Windows

Hiding important files in your EXEs

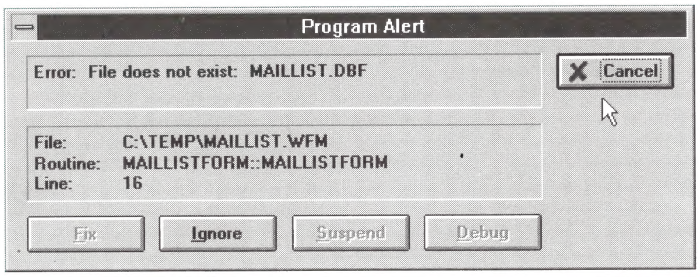
by Keith G. Chuvala

Creating bullet-proof dBASE applications for distribution can be very challenging. No matter how much experience you have or how well you know your clients, it's difficult to anticipate the various error conditions that might occur as users operate your application. In this article, we'll discuss one of the most common problems you'll encounter when you compile and distribute an executable file, and we'll show you a solution.

Problems with deleted files

One class of error that's particularly annoying occurs when a user "cleans up" the hard drive and deletes required files from the directory in which a dBASE application resides. When the user subsequently tries to run the application, the resulting error message, shown in **Figure A**, is nothing short of ugly. If the user presses the Cancel button, the program aborts. Even worse, if the user presses the Ignore button, more error messages result!

Figure A



```
Error:  File does not exist:  MAILLIST.DBF

File:     C:\TEMP\MAILLIST.WFM
Routine:  MAILLISTFORM::MAILLISTFORM
Line:     16
```

When a table is missing, ugliness results.

You can use the dBASE File( ) function to test for the existence of a file before an application attempts to use it. File( ) takes a character string or variable argument specifying which file to look for. It returns .T. if it finds the specified file and .F. if the file doesn't exist. File( ) tests only for single files and doesn't support wildcard characters. By using File( ) to confirm a file's existence, an application can gracefully handle the missing-file error condition, as in the following lines:

```
* Handle a missing data table
*
if .not. file("DATAFILE.DBF")
   MsgBox("The file DATAFILE.DBF is missing;
   can't continue!","An Error has Occurred")
   quit
endif
```

This approach works well, but still renders the application unusable after it displays the error message. What a truly well-behaved application needs is a way to create default files, or at least provide the option to create them, in cases where the user has deleted the original files. The answer lies in the Visual dBASE Compiler.

Packaging data files in EXEs

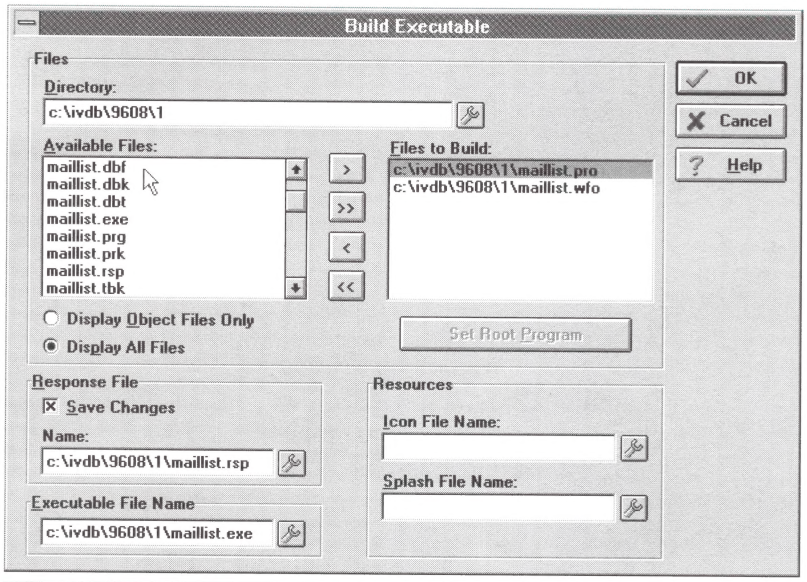
The Visual dBASE compiler inherits from its DOS ancestor an interesting capability

IN THIS ISSUE

that makes it easy for your applications to handle missing files easily. When you create an executable file, you can include *any* kind of file in the EXE.

By default, dBASE's Build Executable dialog box shows only compiled program files. To display all files in the application directory for possible inclusion in the EXE file, click the Display All Files radio button, as shown in Figure B. Add the files you'd like to package inside the EXE file to the Files To Build list, then click the OK button to create the executable file. You'll notice that the size of the EXE file increases accordingly. Files you include in the EXE aren't compressed in any way, so adding a 300-KB bitmap to the file list will bloat the EXE file by 300 KB!

Figure B



Build Executable

Files

Directory: c:\ivdb\9608\1

Available Files:
maillist.dbf
maillist.dbk
maillist.dbt
maillist.exe
maillist.prg
maillist.prk
maillist.rsp
maillist.tbk

Files to Build:
c:\ivdb\9608\1\maillist.pro
c:\ivdb\9608\1\maillist.wfo

( ) Display Object Files Only
(•) Display All Files

Response File
[X] Save Changes
Name: c:\ivdb\9608\1\maillist.rsp

Executable File Name
c:\ivdb\9608\1\maillist.exe

Resources
Icon File Name:
Splash File Name:

Click Display All Files to add non-program files to your EXE.

Restoring from the EXE

Now that you've added the files to the EXE, you can copy them to the disk any time you need fresh copies. This safety net is possible because of extended properties of the File( ) function and the COPY FILE command. Your program can test for the inclusion of a file in the EXE by specifying a second argument in the call to File( ). This second argument can be anything at all; I like to use a string that documents what I'm doing, as illustrated in the following code segment:

```
if file("maillist.dbf","Check in the .EXE file")
   MsgBox("MAILLIST.DBF is contained in the
➥ .EXE", "Success!")
endif
```

The COPY FILE command looks for the file in the EXE first; there's no special syntax or argument to make this happen. So, we can use the File( ) function first to ascertain whether a file exists on disk, and then to check in the EXE file. If necessary, COPY FILE will create a new disk file. Listing A shows a simple example of how you might create such a file at the beginning of an application.

Listing A: *Sample check for a missing file*

```
* Handling missing files at program startup
* Check for data file.  If not there, create a new one
*
if .not. file("maillist.dbf")  && Check the current directory for
   #define Yes 6
   #define YesNoButtons 4
   if MsgBox("Create a new set of data files?",;
"Data files not found on disk!", YesNoButtons) == Yes)
      copy file maillist.dbf to .\maillist.dbf
* Copy the necessary files from the .EXE to disk
      copy file maillist.dbt to .\maillist.dbt
      copy file maillist.mdx to .\maillist.mdx
   else
      MsgBox("The program will exit now.
➥     Please restore the datafiles and try again!",;
         "No data to work with!")
      quit
   endif
else
   use maillist order tag(1) in select()
endif
* The program continues normally....
```

Another bullet dodged!

Packaging default files in your EXE applications can minimize or even eliminate ugly error messages related to missing files. Simply restoring a default file doesn't deal with the real problem—lost or damaged data—or with the events that caused the file or files to be deleted or moved in the first place. However, using this technique in your applications is an important step in creating bullet-proof applications. And, best of all, with File( ) and COPY FILE, it's easy to do! ❖

Creating pop-up menus like the pros

If you've used Visual dBASE, any Microsoft Office application, or any of several other "big" Windows applications, you've probably become pretty accustomed to—maybe even spoiled by—context-sensitive pop-up menus. Windows 95 has established the use of these right-click menus as an interface standard, so we know they're here to stay. The menu display varies, depending on the object the mouse is over when you right-click. This practice creates a fairly intuitive way to perform actions on a given object without requiring extensive multi-tiered window menus.

Visual dBASE uses these nifty pop-up menus throughout the application. Borland added support for pop-up menus to the dBASE language in the transition from dBASE for Windows to Visual dBASE, so you can now add these handy menus to your own applications.

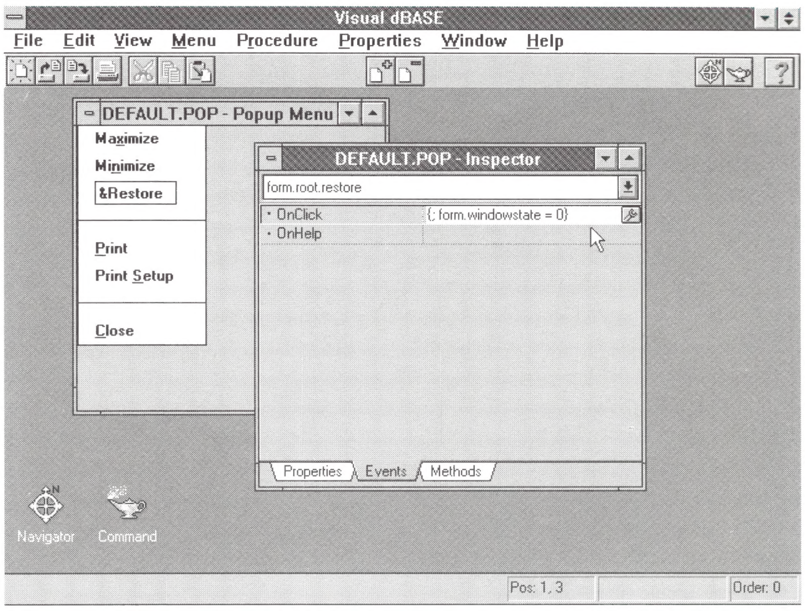
Creating a pop-up menu

You can create pop-up menus much as you would regular dBASE menus, though dBASE saves the pop-up files with the extension POP. **Figure A** shows the design of DEFAULT.POP, a pop-up menu that offers commonly used options you might want to attach to any form you create. You'll find the code for this pop-up in **Listing A**, on page 4. Pop-ups are normally instantiated (that is, a pop-up menu object is created) in a form's OnOpen event and then assigned to the form's PopupMenu property.

Listing B, also on page 4, shows the code for a plain form that contains nothing on it but the pop-up menu. Okay, so it's not an award-winning form in terms of its features, but it efficiently illustrates the steps necessary to add a pop-up menu to a form.

You'll notice that a form's MenuFile has a simple assignment value—the menu's MNU filename—while the PopupMenu property requires the menu to be already instantiated as an object. This seeming inconsistency actually works to your advantage once you graduate to more than one pop-up menu per form. **Figure B** shows DEFAULT.WFM with the pop-up menu active.

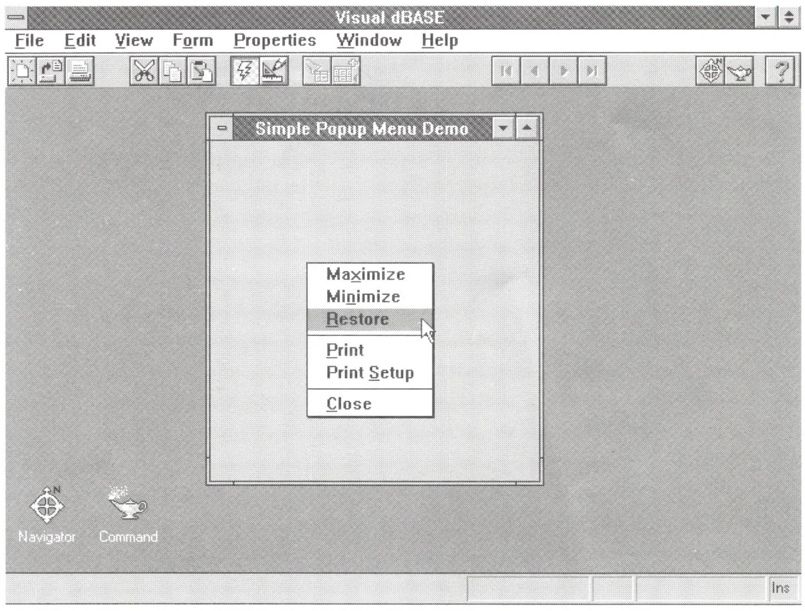
Figure A



form.root.restore
OnClick {; form.windowstate = 0}
OnHelp

You can create a pop-up menu with the Menu Designer.

Figure B



Here's a simple form with the basic pop-up menu, DEFAULT.WFM, in action.

One menu per customer, please

At first glance, the PopupMenu property seems similar to a MenuFile property: You set it once and forget about it. True, you can do that. The DEFAULT.POP menu in our example is suitable for almost any form. But in most applications, you'll want your pop-ups to change according to the object the mouse is currently over.

To add this functionality, you can use the OnMouseMove event handler, which is available as part of all visible interface objects in Visual dBASE. OnMouseMove's event handler executes any time the mouse moves over an object, as well as every time the mouse moves by one pixel or more while over that object.

Notice that this execution pattern is quite different from that of OnGotFocus's event handler, which fires only once, when the object gets focus. OnMouseMove doesn't even have a focus requirement—the mouse pointer just has to pass over any visible part of the object to trigger the event handler.

Listing A: *The DEFAULT.POP: A generic pop-up menu*

```
** END HEADER * do not remove this line*
* Generated on 05/12/96

*
Parameter FormObj,PopupName
NEW DEFAULTPOPUP(FormObj,PopupName)
CLASS DEFAULTPOPUP(FormObj,PopupName)
➥ OF POPUP(FormObj,PopupName)
   this.Left = 0
   this.Top = 0
   this.TrackRight = .T.

   DEFINE MENU MAXIMIZE OF THIS;
       PROPERTY;
         Text "Ma&ximize",;
         OnClick {; form.windowstate = 2}

   DEFINE MENU MINIMIZE OF THIS;
       PROPERTY;
         Text "Mi&nimize",;
         OnClick {; form.windowstate = 1}

   DEFINE MENU RESTORE OF THIS;
       PROPERTY;
         Text "&Restore",;
         OnClick {; form.windowstate = 0}

   DEFINE MENU MENU368 OF THIS;
       PROPERTY;
         Text "",;
         Separator .T.

   DEFINE MENU PRINT OF THIS;
       PROPERTY;
         Text "&Print",;
         OnClick {; form.print()}

   DEFINE MENU PRINT_SETUP OF THIS;
       PROPERTY;
         Text "Print &Setup",;
         OnClick {; chooseprinter()}

   DEFINE MENU MENU476 OF THIS;
       PROPERTY;
         Text "",;
         Separator .T.

   DEFINE MENU CLOSE OF THIS;
       PROPERTY;
         Text "&Close",;
         OnClick {; form.close()}

ENDCLASS
```

Listing B: *DEFAULT.WFM: A simple form with a pop-up menu*

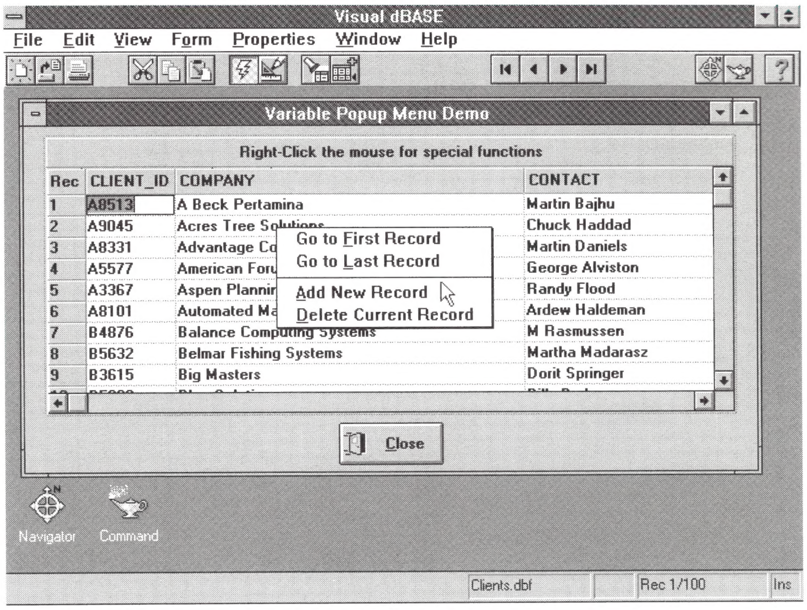
```
** END HEADER * do not remove this line*
* Generated on 05/12/96
*
parameter bModal
local f
f = new DEFAULTFORM()
if (bModal)
   f.mdi = .F. && ensure not MDI
   f.ReadModal()
else
   f.Open()
endif
CLASS DEFAULTFORM OF FORM
   this.OnOpen = CLASS::FORM_ONOPEN
   this.Text   = "Simple Popup Menu Demo"
   this.Height = 16
   this.Width  = 44
   this.Left   = 10
   this.Top    = 3

   Procedure FORM_OnOpen
      IF TYPE("Form.PopupMenu") # "O"
       DO default.pop with form,"DefaultPopup"
       form.PopupMenu = form.DefaultPopup
      ENDIF
ENDCLASS
```

Listing C defines a form that uses DEFAULT.POP and adds another pop-up menu, BROWSE.POP, that opens when the user positions the mouse over a Browse object on the form. This new pop-up menu (see Listing D, on page 6, for its source code) must also be instantiated; the form's OnOpen event handler, FORM_OnOpen, handles that process for you. (Note that each PopupMenu object must have a unique name.) Figure C shows BROWSE.POP activated over the Browse object. As soon as the mouse moves off the Browse object, DEFAULT.POP becomes the active pop-up menu.

The BROWSE1_OnMouseMove procedure and the FORM_OnMouseMove procedure do all the work, if you can call it that! Note that each procedure simply reassigns the form's PopupMenu property to the appropriate pop-up menu.

To add additional pop-up menus to your forms, you'll follow a similar three-step

Figure C



Right-Click the mouse for special functions

Rec	CLIENT_ID	COMPANY	CONTACT
1	A8513	A Beck Pertamina	Martin Bajhu
2	A9045	Acres Tree So	Chuck Haddad
3	A8331	Advantage Co	Martin Daniels
4	A5577	American Foru	George Alviston
5	A3367	Aspen Plannir	Randy Flood
6	A8101	Automated Ma	Ardew Haldeman
7	B4876	Balance Computing Systems	M Rasmussen
8	B5632	Belmar Fishing Systems	Martha Madarasz
9	B3615	Big Masters	Dorit Springer

We activated BROWSE.POP by right-clicking over a Browse object.

Listing C: *A form with multiple pop-up menus*

```
** END HEADER * do not remove this line*
* Generated on 05/13/96
*
parameter bModal
local f
f = new RMOUSEFORM()
if (bModal)
   f.mdi = .F. && ensure not MDI
   f.ReadModal()
else
   f.Open()
endif
CLASS RMOUSEFORM OF FORM
   this.OnOpen = CLASS::FORM_ONOPEN
   this.Left = 2
   this.Top = 2
   this.Text = "Variable Popup Menu Demo"
   this.Height = 16.5
   this.Width = 98
   this.OnMouseMove = CLASS::FORM_ONMOUSEMOVE
   this.View = "CLIENTS.DBF"

   DEFINE BROWSE BROWSE1 OF THIS;
       PROPERTY;
         Left 3,;
         Top 0.5,;
         CUATab .T.,;
         FontBold .T.,;
         Text "Right-Click the mouse for special functions",;
         Height 13.0293,;
         Width 92,;
         OnMouseMove CLASS::BROWSE1_ONMOUSEMOVE

   DEFINE PUSHBUTTON PB_CLOSE OF THIS;
       PROPERTY;
         Left 42,;
         Top 14,;
         Text "&Close",;
         Height 2,;
         Group .T.,;
         Width 14,;
         UpBitmap "RESOURCE #1005",;
         OnClick {; form.close()}

   Procedure BROWSE1_OnMouseMove(flags, col, row)
      form.popupmenu=form.BrowsePopup

   Procedure Form_OnMouseMove(flags, col, row)
      form.popupmenu=form.DefaultPopup

   Procedure FORM_OnOpen
      IF TYPE("Form.PopupMenu") # "O"
         DO default.pop with form,"DefaultPopup"
         form.PopupMenu = form.DefaultPopup
      ENDIF
      IF TYPE("Form.BrowsePopup") # "O"
         DO browse.pop with form,"BrowsePopup"
      ENDIF
ENDCLASS
```

process. First, use the menu designer to create the POP file. Next, instantiate a unique variable for the pop-up menu in the form's OnOpen event handler. Finally, you'll add an OnMouseMove event handler for each object that you want to activate the new pop-up menu.

Get popping!

Pop-up menus add a nice touch to your applications, making them more intuitive to the user and more professional in appearance. Many of your customers and clients probably use Windows 95; thus they expect context-sensitive menu functionality and will likely notice its absence.

Fortunately, Visual dBASE makes it easy for you to create and add pop-up menus to your applications. Now your programs can offer the familiar interface enhancements that users find in the big, commercial programs. ❖

Listing D: *BROWSE.POP: A pop-up menu for Browse objects*

```
** END HEADER * do not remove this line*
* Generated on 05/13/96
*
Parameter FormObj,PopupName
NEW BROWSEPOPUP(FormObj,PopupName)
CLASS BROWSEPOPUP(FormObj,PopupName) OF POPUP(FormObj,PopupName)
   this.Left = 0
   this.Top = 0
   this.TrackRight = .T.

   DEFINE MENU GO_TO_FIRST_RECORD OF THIS;
       PROPERTY;
         Text "Go to &First Record",;
         OnClick {; go top}

   DEFINE MENU GO_TO_LAST_RECORD OF THIS;
       PROPERTY;
         Text "Go to &Last Record",;
         OnClick {; go bottom}

   DEFINE MENU MENU137 OF THIS;
       PROPERTY;
         Text "",;
         Separator .T.

   DEFINE MENU ADD_NEW_RECORD OF THIS;
       PROPERTY;
         Text "&Add New Record",;
         OnClick {; form.beginappend()}

   DEFINE MENU DELETE_CURRENT_RECORD OF THIS;
       PROPERTY;
         Text "&Delete Current Record",;
         OnClick {; if msgbox("Mark the current record for
➥        deletion?","Are you sure?",4)=6;
➥        delete next 1; endif}

ENDCLASS
```

PROGRAMMING TECHNIQUE

Creating temporary (or any!) tables on the fly

by Keith G. Chuvala

Complex applications that handle large amounts of data often require the use of temporary tables, and dBASE provides a useful feature in its ability to create new tables on the fly. For example, you may sometimes find that it's easier to write information to a temporary DBF table to produce a sophisticated report than it is to wrangle with Crystal Reports for dBASE. In this article, we'll discuss some of the ways dBASE lets you create tables on the fly, and we'll explore how you can use these techniques in your applications.

You can create tables the old-fashioned way, since Visual dBASE supports the dBASE/DOS command

```
CREATE <table_name> FROM
    <structure_extended_table>
```

This command works fine, but I hate it!

The <structure_extended_table> needs to be yet another DBF or DB table containing specific fields, usually using the command

```
COPY TO <table_name> STRUCTURE EXTENDED
```

Overall, this approach is clunky at best. Furthermore, if a user happens to delete the CREATE command's source table from the disk during one of those surely-I-don't-need-that-file moods, the command becomes useless.

SQL to the rescue, sort of

The SQL command CREATE TABLE creates a table on the fly, and it doesn't require an additional file like CREATE FROM. There's one problem, though–it's SQL, not dBASE—and, well, it's kind of weird if you're not used to SQL. The command's syntax is straightforward:

```
CREATE TABLE <table_name> (<field_name>
    <data_type> [,<field_name>
    <data_type>...] )
```

What's so weird about that? You must specify the <data_type> for each field as an SQL data type. Table A lists a number of SQL data types and their dBASE equivalents. Note that CREATE TABLE can create Paradox tables as well as dBASE tables, so the list of SQL data types isn't exhaustive. But, the list is complete as far as mapping SQL data to Visual dBASE data is concerned. Both SQL types—NUMERIC and FLOAT—take two arguments in parentheses: the width and the number of decimal places for the field. Similarly, the CHARACTER type takes a single argument in parentheses, which specifies the width of the character field.

Table A: *SQL and dBASE data types*

SQL Data Type	dBASE Equivalent
SMALLINT	Numeric (6,0)
INTEGER	Numeric (11,0)
NUMERIC(x,y)	Numeric (x,y)
FLOAT(x,y)	Float (x,y)
MONEY	Float (20,4)
CHARACTER(n)	Character
DATE	Date
BOOLEAN	Logical
BLOB(n,1)	Memo
BLOB(n,2)	Binary
BLOB(n,4)	OLE

Let's take CREATE TABLE for a spin. In the Command window, enter the command

```
CREATE TABLE MYCARS (MAKE CHARACTER(12),
➥      MODEL CHARACTER(20),
➥      MODYEAR NUMERIC(4,0),
➥      ODOMETER NUMERIC(8,1),
➥      PURCHASED DATE, EMISSIONS BOOLEAN)
```

You've just created MYCARS.DBF. Let's look at the table's structure. To do so, enter the commands

```
USE MYCARS
DISPLAY STRUCTURE
```

In the Command window's Results pane, you'll see the following:

```
Structure for table    C:\IVDB\9608\3\MYCARS.DBF
Table type             DBASE
Number of records      0
Last update            06/04/96
Field  Field Name                 Type          Length  Dec  Index
    1  MAKE                       CHARACTER         12            N
    2  MODEL                      CHARACTER         20            N
    3  MODYEAR                    NUMERIC            4            N
    4  ODOMETER                   NUMERIC            8    1       N
    5  PURCHASED                  DATE               8            N
    6  EMISSIONS                  LOGICAL            1            N
** Total **                                         54
```

So far so good, but...

The SQL approach works well, doesn't it? CREATE TABLE has a few shortcomings, though. First, it doesn't provide a way to create indexes. Second, CREATE TABLE won't overwrite an existing table with the same name as the one specified in the command. SET SAFETY OFF, which normally dictates that a file should be overwritten without warning, has no effect on CREATE TABLE. In addition, you get a rather forbidding message when the table already exists, as shown in Figure A on page 8—which is certainly not a message you want your application's users to see!

I use temporary and other on-the-fly tables often enough that I created a function that takes advantage of CREATE TABLE's features while working around its limitations. Listing A contains the result:

MkTable( ). MkTable( ) takes two arguments—the name of the table to create, and an array containing the new file's structure.

You may think I'm borrowing trouble using an array, since you could argue that passing an array as an argument to MkTable( ) makes the function difficult to use. And it would be, if we were using dBASE for DOS. But Visual dBASE supports literal arrays. You might have encountered these in the Form Designer and not even realized what they were. A *literal array* is written with the array elements enclosed in curly braces: { and }.

For example, try entering the following line in the Command window:

```
x = {"first",2,3.0,date(),"five"}
```

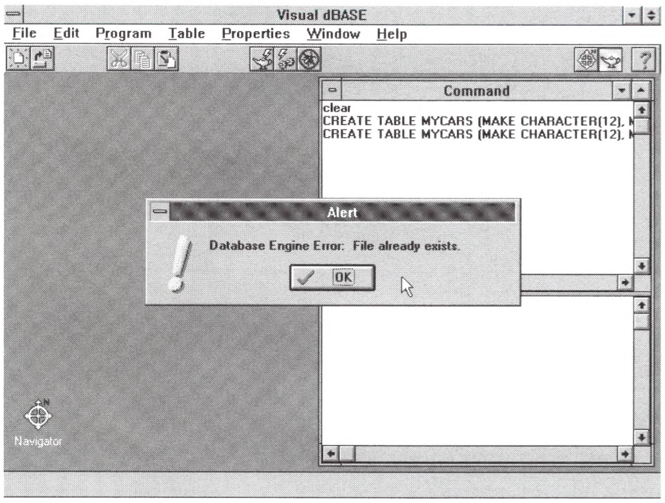
You've just created an array called X that has five elements. Type in the command DISPLAY MEMORY to see the following results:

```
User Memory Variables

X            Pub   A  [5]
         [    1] C  "first"
         [    2] N            2.00 (2.00000000000000)
         [    3] N            3.00 (3.00000000000000)
         [    4] D  05/24/96
         [    5] C  "five"
     1 variables defined,      25 bytes used
   499 variables available,  4071 bytes available
```

You can pass an array to a function in the same way. The second parameter to MkTable( ) can take the form

```
{"Field1name, Type, Length, Decimals,
➥    IndexType", "Field1name, Type, Length,
➥    Decimals, IndexType"}
```

Figure A



```
clear
CREATE TABLE MYCARS (MAKE CHARACTER(12), N
CREATE TABLE MYCARS (MAKE CHARACTER(12), N
```

Database Engine Error: File already exists.

These unpleasant results occur when you run the same CREATE TABLE command more than once.

Each string in the array contains two or more parts, with the parts separated by commas. The *Field1name* component must be a valid dBASE field name. *Type* is a single-character designator for the field type: C for character, N for numeric, F for float, D for date, L for logical, M for memo, O for OLE, or B for Binary.

Length is the field width for Character, Numeric, and Float fields. *Decimals* specifies the number of decimal places to reserve for Numeric and Float fields. *IndexType* is a single character expression: A for an ascending index tag, or D for a descending index tag.

Using MkTable( )

Let's use MkTable( ) to create a table called TEMPFILE.DBF with the following structure:

```
Field  Field Name   Type       Length   Dec    Index
    1  LASTNAME     CHARACTER  15              N
    2  FIRSTNAME    CHARACTER  12              N
    3  SALARY       NUMERIC    8        2      Y
    4  DATE_HIRED   DATE       8               Y
```

The command would take the form

```
lOK = MkTable("TEMPFILE", {"Lastname,C,15",
➥    "Firstname,C,12", "Salary,N,8,2,A",
➥    "Date_Hired,D,D"})
```

If the file was created successfully, lOK will be set to .T.; otherwise, it will be .F.. With this approach, your program can test the success of the operation.

Conclusion

With the CREATE TABLE command and the MkTable( ) program, your applications can create tables of almost any structure on demand. Use these tools to make your programming life easier! ❖

Listing A: *MKTABLE.PRG*

```
*
* MkTable(cTableName,aStructure): Create a new table on the fly
*
* Arguments: cTableName - a string with the table name
*            aStructure - an array of strings in the format:
*                         FieldName,Type,Width,Decimals,Index
*
function mktable
   parameter cTable,aStruct,lOverwrite
   private cCommand,cSQL,cTableQ,aTags
   local i,cField,cName,cType,cWidth,cDec,cMDX,cAlias
   aTags = new array()

   * Make sure we've got the two reqiured parameters
   if (type("cTable") # "C") .or. (type("aStruct") # "A")
      msgbox('MakeTable("tablename",{array})',"Argument error!")
      return .f.
   endif

   * Default to no overwrite if not specified
   if type("lOverwrite") # "L"
      lOverwrite = .f.
   endif

   * Make the file name pretty
   cTable = trim(ltrim(upper(cTable)))
   if right(cTable,4) # ".DBF"
      cTable = cTable + ".DBF"
   endif

   * Delete existing file if called for
   if file(cTable)
      #define YesNoOpts 4
      #define YesButton 6
      if .not. lOverwrite
         lOverwrite = msgbox("Do you want to overwrite it?", ;
                             ctable + " already exists",;
                             YesNoOpts) == YesButton
      endif
      if lOverwrite
         delete file (cTable)
      else
         return .f.
      endif
   endif

   cCommand = " ("  && This holds all parameters for CREATE TABLE
                    && cSQL is used for each individual field
   for i = 1 to aStruct.Size
      cSave = upper(aStruct[i])
      cfield = cSave
      cName = trim(ltrim(NextToken(cField)))
      cType = trim(ltrim(NextToken(cField)))
      do case
         case cType = "C"
            cWidth  = NextToken(cField)
            cMDX    = NextToken(cField)
            if .not. isblank(cMDX)
               aTags.add(cName)
               aTags.add(cMDX)
            endif
            cSQL = cName + " CHARACTER(" + cWidth + ")"
         case cType = "N"
            cWidth = NextToken(cField)
            cDec   = NextToken(cField)
            cMDX   = NextToken(cField)
            if .not. isblank(cMDX)
               aTags.add(cName)
               aTags.add(cMDX)
            endif
            cSQL = cName + " NUMERIC(" + cWidth + "," + cDec + ")"
         case cType = "F"
            cWidth = NextToken(cField)
            cDec   = NextToken(cField)
            cMDX   = NextToken(cField)
            if .not. isblank(cMDX)
               aTags.add(cName)
               aTags.add(cMDX)
            endif
            cSQL = cName + " FLOAT(" + cWidth + "," + cDec + ")"
         case cType = "D"
            cMDX = NextToken(cField)
            if .not. isblank(cMDX)
               aTags.add(cName)
               aTags.add(cMDX)
            endif
            cSQL = cName + " DATE"
         case cType = "L"
            cSQL = cName + " BOOLEAN"
         case cType = "M"
            cSQL = cName + " BLOB(10,1)"
         case cType = "B"
            cSQL = cName + " BLOB(10,2)"
         case cType = "O"
            cSQL = cName + " BLOB(10,4)"
         otherwise
            msgbox(cField+" is invalid!","Field format error")
      endcase
      if len(cCommand) > 2
         cCommand = cCommand + ","
      endif
      cCommand = cCommand + cSQL
   next

   cCommand = cCommand + ")"
   cTableQ = '"'+cTable+'"'
   CREATE TABLE &cTableQ. &cCommand

   *
   * Create indexes (if any)
   *
   cAlias = alias()
   select select()
   use (cTable) excl
   for i = 1 to aTags.Size Step 2
      cTag = aTags[i]
      cOrder = aTags[i+1]
      do case
         case cOrder $ "YA"
            index on &cTag. tag &cTag.
         case cOrder $ "DZ"
            index on &cTag. tag &cTag. descending
         otherwise
            msgbox(cOrder + " is invalid.  Use A or D",;
                   "Invalid sort order!")
      endcase
   next
   use
   if len(trim(cAlias)) > 0
      select (cAlias)
   endif
return .t.

function NextToken

   parameter cString
   local nComma,cRetVal
   nComma = at(",",cString)
   if nComma > 0
      cRetVal = left(cString,nComma - 1)
      cString = substr(cString,nComma+1)
   else
      cRetVal = cString
      cString = ""
   endif
return cRetVal
```

Tipping the Scale(font)s

One of the neat things about Windows—for users—is the varied array of video resolutions available. At the same time, one of the bothersome things about Windows—for developers—is the varied array of video resolutions available. Long gone are the days when we software developers could make assumptions about the kinds of video cards our software would have to support. Now, we have to be smarter than the machines.

Let's make this perfectly clear

The Windows operating systems and Windows applications do their best to be device *independent*. The hardware shouldn't matter to your application. That is, a Windows-based program should be able to run on any video card supported by a Windows driver, and at any resolution supported by a Windows driver.

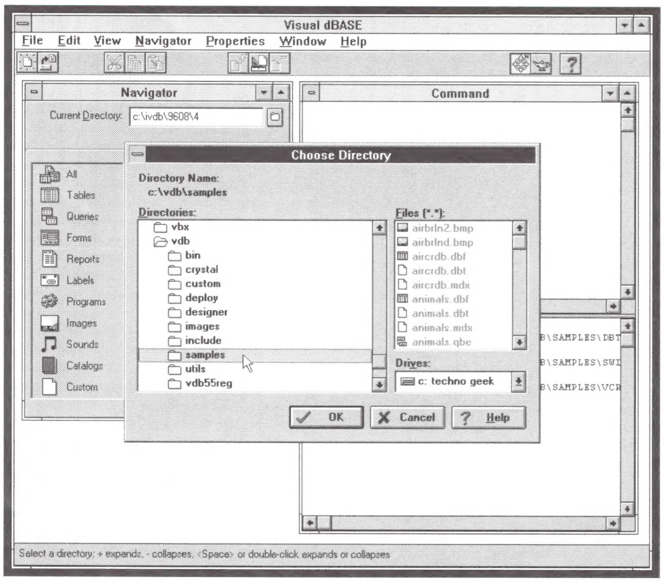
Video resolution is measured in pixels. A *pixel* is the single smallest dot of light that can be displayed on the computer's monitor. A standard VGA card runs a maximum resolution of 640 pixels horizontally and 480 pixels vertically. Most video cards sold for the past few years have far exceeded the capabilities of standard VGA, supporting resolutions as high as 1280 x 1024—and sometimes even higher!

Smaller monitors usually run at lower resolutions, and larger monitors run at higher resolutions. The higher the resolution, the more information you can display on the screen, but also the smaller each pixel—and therefore each item on the screen—becomes.

Deciding which resolution to use is generally a matter of choosing the highest value you can use comfortably on the monitor attached to the system. For example, I have a laptop computer running at 640- x 480-pixel resolution, a desktop machine at home with a 14-inch monitor that I run comfortably at 800 x 600 pixels, and a machine at the office with a gorgeous, 17-inch monitor that I run at 1024 x 768 pixels.

Creating applications that can resize their forms and other visual elements to match the resolution of the screen can be tricky. Fortunately, Visual dBASE forms have a couple of properties to make the job easier: ScaleFontName and ScaleFontSize.

Figure A



Select a directory: + expands, - collapses, <Space> or double-click expands or collapses

Use the Navigator to select the Visual dBASE SAMPLES directory.

The ScaleFont properties

Before we get into the specifics of using ScaleFontName and ScaleFontSize, let's talk about how Visual dBASE measures elements. Working in pixels can be frustrating, because they don't directly correspond to any part of an application.

The sizes of Visual dBASE objects, such as forms and pushbuttons, aren't measured in pixels. Instead, dBASE measures the height and width of each element in character boxes. A *character box* is approximately the amount of space that a character occupies onscreen when you've chosen a particular font at a particular size. It's sometimes easier to think of these boxes in terms of rows (vertical resolution) and columns (horizontal resolution). So, a form with a Height property set to 20 and a Width property set to 60 is approximately 20 rows (or characters) high and 60 columns wide.

The grid that Visual dBASE uses to keep track of these rows and columns is called the *coordinate plane*. The ScaleFontName and ScaleFontSize properties specify the font used to calculate the size of each character box on a form. If you change the font and/or size used, you change the number of rows and columns that can fit on a form.

Try it!

Let's use one of the sample forms included with Visual dBASE to illustrate the effect these two properties have on a form. In the Navigator, select the directory that contains the Visual dBASE sample files, as shown in Figure A. On my system, it's C:\VDB\SAMPLES. If you installed Visual dBASE in the default location, the directory will probably be C:\VISUALDB\SAMPLES.

In the Command window, type the following commands to create a new form object from the ANIMALS.WFM file:

```
SET PROCEDURE TO ANIMALS.WFM ADDITIVE

F = NEW ANIMALSFORM( )
F.OPEN( )
```

The first line makes the form specified in that file available to us from the Command window. The second line creates a form called F that we can play with, and the third line opens the form, as shown in Figure B.

ScaleFontName's default value is MS SanSerif, which is not a scalable font. Let's change the ScaleFontName property to Arial by typing in the Command window

```
F.ScaleFontName="Arial"
```

Did you notice a slight change in the size of the form? The character box size of the Arial font is slightly smaller than that of the MS SanSerif font, and the form adjusts itself accordingly.

Now let's play with ScaleFontSize. The default is 10. Let's increase the size and watch the effect. In the Command window, enter

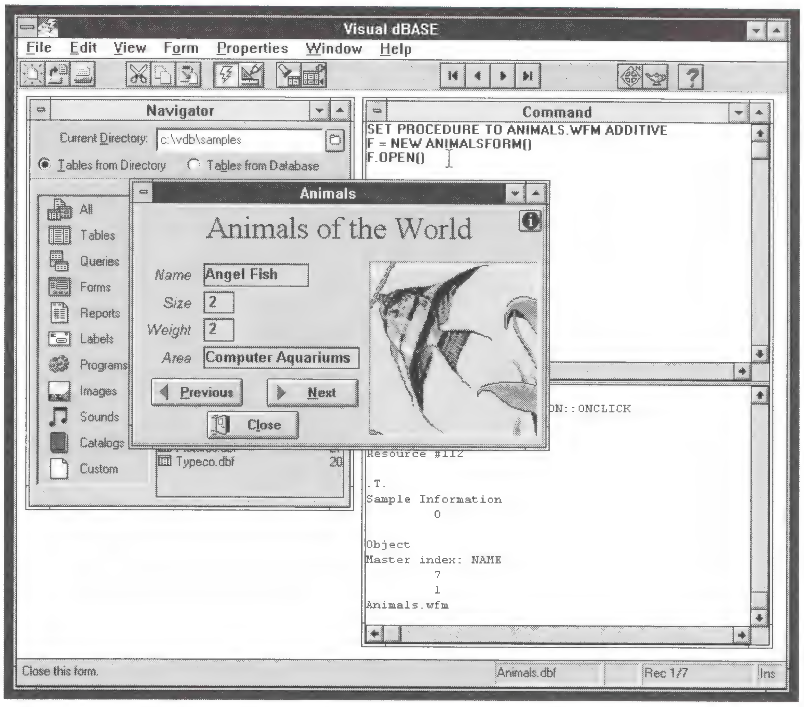
```
F.ScaleFontSize=14
```

Wow! Notice how everything got larger? The form itself, the buttons, entry fields, text objects—and even the image control containing the picture of the Angel Fish—grew, as shown in Figure C. Try other ScaleFontSize settings and watch the form rescale itself.

It's a start

For optimal benefit, the fonts you use for your pushbuttons, entry fields, and so on,

Figure B



```
SET PROCEDURE TO ANIMALS.WFM ADDITIVE
F = NEW ANIMALSFORM()
F.OPEN()
```

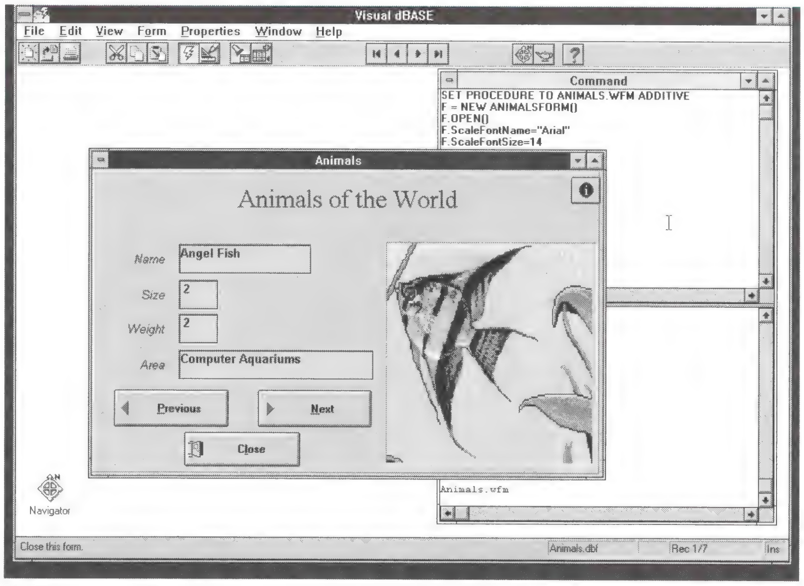
Name: Angel Fish
Size: 2
Weight: 2
Area: Computer Aquariums

```
ON::ONCLICK
Resource #112
.T.
Sample Information
        0

Object
Master index: NAME
        7
        1
Animals.wfm
```

Close this form. | Animals.dbf | Rec 1/7 | Ins

We opened this Animals form from the Command window.

Figure C



```
SET PROCEDURE TO ANIMALS.WFM ADDITIVE
F = NEW ANIMALSFORM()
F.OPEN()
F.ScaleFontName="Arial"
F.ScaleFontSize=14
```

Name: Angel Fish
Size: 2
Weight: 2
Area: Computer Aquariums

```
Animals.wfm
```

Close this form. | Animals.dbf | Rec 1/7 | Ins

Our Angel Fish is putting on weight!

should be scalable. Arial works well, but any TrueType, Type 1, or other scalable font will behave similarly.

There's much more to be explored on this topic. In a future issue, we'll examine other ways to use ScaleFont, and we'll present additional resizing tricks and techniques. ❖

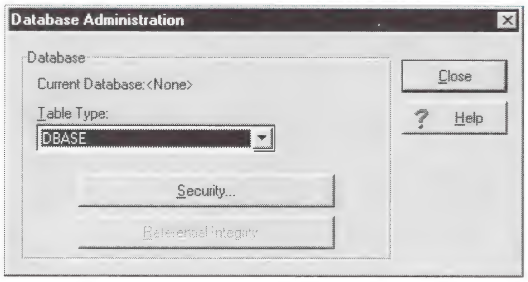
Implementing password-protected login access to Visual dBASE

Please note this advance warning: Use extreme caution in trying the technique we describe in this article. If you forget the password you create for yourself, you may be forced to re-install the program to regain access.

If you use Visual dBASE to maintain confidential databases, you can protect sensitive information from roving eyes by copying an important file onto a floppy disk, erasing it from the hard drive, taking the disk home with you after work every day, and restoring the file every morning. Even using that method of security with your files comes with a fair amount of risk—after all, you'd be in big trouble if you lost or damaged the diskette.

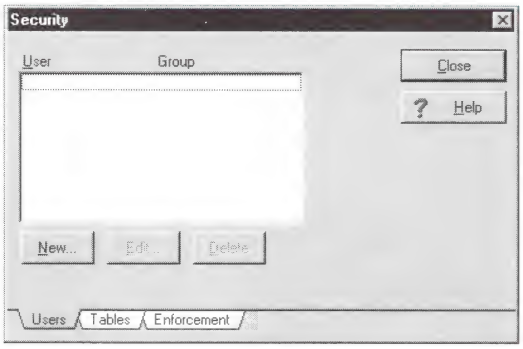
Fortunately, Visual dBASE provides a number of built-in tools you can use to keep your files safe. In this article, we'll take a look at Visual dBASE's first level of security—login access—by showing you how to password protect entry to Visual dBASE itself. (In a future issue, we'll introduce you to the other two methods of securing your data: data encryption, and table and field security.)

Figure A



You'll use this dialog box to implement password protection for Visual dBASE.

Figure B



This dialog box lets you access your system's security features.

Keeping the rascals out

Suppose you've installed Visual dBASE on a computer that's located in a public area, and you want to make sure no one but you can run the program. To do so, you define three login parameters for yourself: a group name, a login name, and a password. Then, the next time you try to start Visual dBASE, a dialog box prompting you to enter those values will appear. If you fail to enter the correct login values, Visual dBASE simply won't run.

To begin, start Visual dBASE and choose Database Administration... from the File menu. When you do, the Database Administration dialog box appears, as shown in Figure A.

The first item in the Database Administration dialog box tells Visual dBASE what types of databases to display in the Navigator. The type of table can affect whether security features are available. Typically, you'll accept the default value of DBASE for this option.

Now, click the Security... button. The first time you do, the Administrator Password dialog box appears. Type the password you'll use as the database administrator. Once you define the administrator password, keep a hard copy of that password in a secure place.

Note: *There is no way to retrieve this password from the system.* After the first use, the Administrator Password dialog box controls access to the security features, and it will appear each time you click the Security... button. You can change the security system only if you first supply the administrator password.

You still have a little more work to do to prevent unwelcome access to Visual dBASE. After you enter the administrator password, the Security dialog box appears, as shown in Figure B.

Now, make sure the Users tab is selected, and then click the New... button. When you do, the New User dialog box appears. The User box lets you specify the login name for a user. Enter a name from one to eight alphanumeric characters in length. (Visual dBASE converts the entry to uppercase.) You can enter up to 8 characters for the Group name and up to 16 characters for the password, including spaces.

When you define the password, you can enter either uppercase or lowercase letters. The password isn't case-sensitive: You just have to remember how to spell it.

Visual dBASE also lets you enter spaces in the password—both when you define it and when you enter it to access the program. However, you'll discover that Visual dBASE ignores the spaces. That is, whether or not the original password you define contains spaces, you can include or omit spaces when you type the login password—Visual dBASE considers only the non-space alphanumeric characters you type.

As an option, you can include the full name of the user whose profile you're defining. At this point, you can accept the default value of 1 for the Access Level; we'll talk more about that option later. Figure C shows a sample user profile.

Next, you must tell Visual dBASE when to restrict access to qualified users. To do so, click the Enforcement tab and the When Loading Visual dBASE radio button. Then, close all the open dialog boxes and exit Visual dBASE. The next time you load Visual dBASE, you'll see the login dialog box. (This dialog box makes no mention of the Access Level you set when you defined the login information for this user; that's because the Access Level becomes important only when you implement table—or table and field—security measures.)

If you click the Visual dBASE Login dialog box's Cancel button, the program will return to your operating system. If you enter an incorrect value in any of the fields, the system will issue a warning message. For instance, if you type *DABULL* instead of *DABULLS* in the Group field, Visual dBASE will display an error message.

If you attempt to gain access three times without providing the correct login information, the program will fail to load and will return you to the operating system. You'll have to double-click the program icon again if you want to try logging in once more.

Disabling login protection

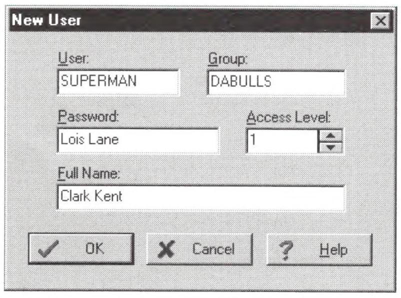
Now that you know how to prevent unwanted users from gaining access to Visual dBASE, you probably want to know how to disable such protection. Doing so is easy, as long as you remember your administrator password. Just choose Database Administration... from the File menu, click the Security... button, and enter that all-important password.

Then, when the Security dialog box appears, you can delete the user profile by clicking the Delete button. Or, you can leave the user profile intact and simply change the way Visual dBASE enforces password protection. Specifically, you'd select the Enforcement tab and select the When Loading An Encrypted Table radio button instead of the When Loading Visual dBASE radio button.

Conclusion

In this article, we showed you how Visual dBASE lets you control access to its environment. In future issues, we'll show you a similar approach to protecting data by encrypting a table's contents or by restricting a user's access to a table or to a field within a table. ❖

Figure C



User: SUPERMAN
Group: DABULLS
Password: Lois Lane
Access Level: 1
Full Name: Clark Kent

Define login access privileges for your users in this dialog box.

Protect your properties

If you browse through the source code for the custom controls included with Visual dBASE, you'll notice a new use for the word PROTECT. PROTECT is known to many dBASE programmers as the password and table security system introduced to dBASE for DOS back in the 1980's. Now there's a new spin on PROTECT that also has to do with security, though in a very different way.

A class background

Custom Controls should be constructed so they function as "black boxes"—as self-contained and self-reliant as possible. The object-oriented programming concept known as *encapsulation* provides a mechanism whereby we can include both code and data in a single unit that we call an "object." We create the definitions of these objects using the now-familiar construct CLASS…ENDCLASS.

Why protect properties?

Often the properties you create inside a custom class are used only by the procedures and functions within that class. dBASE's default behavior, however, is such that any code can change any custom property in any class, whether the change makes sense or not! So, good OOP design calls for a mechanism to—you guessed it—*protect* properties from improper modification from outside the class.

Listing A, DBFBROW.PRG, contains the code for a custom class that creates a new session and triggers a BROWSE window with a specific title. You call it like this:

```
set procedure to dbfbrow
x = ;
new dbfbrow("c:\vdb\samples\customer","Customers
➥ on File")
```

Or, you can create the object first and set the TableName property in a separate command:

```
set procedure to dbfbrow
x = new dbfbrow()
x.SetTableName("c:\vdb\samples\customer")
```

You can see the results in **Figure A**. As long as you follow the rules, this command works great. If you don't, the object is easy to break, too!

Listing A: *DBFBROW.PRG with unprotected properties*

```
* DBFBROW: Create an on-the-fly BROWSE
*
* Usage:  SET PROCDURE TO DBFBROW
*         x = NEW DBFBROW("table_name","Window Title")
*
* CLASS::SetTableName() can be called to set or reset the table name
*
* NOTE!  Custom properties are NOT protected;
* use care not to change their type!

CLASS DBFBROW
   parameters cTable, cTitle
   this.tablename = iif(type("cTable") = "C",cTable,"")
   this.title     = iif(type("cTitle") = "C",cTitle,"Table View")
   this.browsing  = .f.

   PROTECT tablename,title,browsing

   if .not. isblank(this.tablename)
      this.show()
   endif

   procedure SetTableName
      parameter cTable
      if type("cTable") = "C"
         this.tablename = cTable
      endif
   return

   procedure SetTitle
      parameter cTitle
      if type("cTitle") = "C"
         this.title = cTitle
      endif
   return

   procedure Show
      if .not. isblank(this.tablename)
         if .not. this.browsing
            create session
            use (this.tablename)
         else
            use
         endif
         this.browsing = .t.
         if .not. isblank(tag(1))
            set order to (tag(1))
         endif
         go top
         browse title (this.title)
      endif
   return

ENDCLASS
```

Where's the real problem?

Let's try an experiment. Suppose you change the TableName property to the wrong data type:

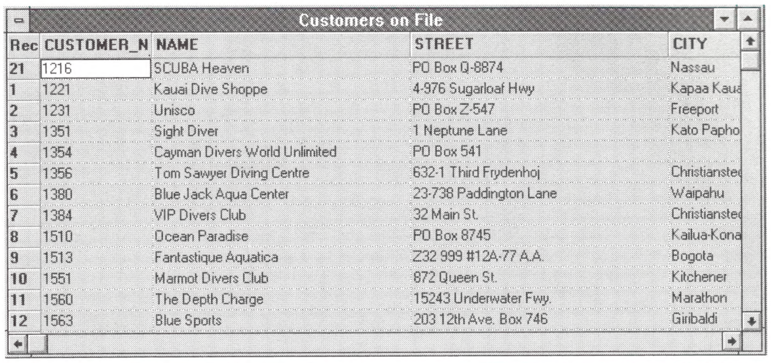
```
* PROBLEM.PRG - breaks the DBFBROW class
set procedure to dbfbrow
x = new dbfbrow()
x.TableName = .f.
x.Show()
```

Setting the TableName property to .F. is, well, dumb. But it's also entirely legal syntactically. So, this code would compile just fine. When you run the program, however, you'll get an error, as shown in Figure B.

Of course, this kind of error is going to happen no matter how careful you are, especially in large projects. The real (and avoidable) problem, then, is that the error message tells us where the program stopped in the DBFBROW class, which is *not* where the actual programming error is situated! You need to know the location of the code that incorrectly changed the data type of the TableName property to begin with.

Here's where PROTECT can prevent runtime errors. Listing B, on page 16, shows another version of DBFBROW.PRG. The only difference between this one and the previous class definition is the addition of PROTECT to prevent unwanted modification of the TableName, Title, and Browsing properties from outside of the class. Now let's run the problematic code again, and look at the change in the error

Figure A

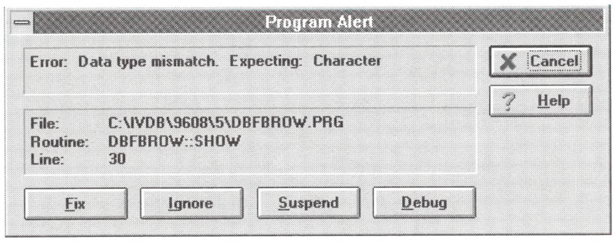


Customers on File

Rec	CUSTOMER_N	NAME	STREET	CITY
21	1216	SCUBA Heaven	PO Box Q-8874	Nassau
1	1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Kapaa Kaua
2	1231	Unisco	PO Box Z-547	Freeport
3	1351	Sight Diver	1 Neptune Lane	Kato Papho
4	1354	Cayman Divers World Unlimited	PO Box 541	
5	1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	Christianste
6	1380	Blue Jack Aqua Center	23-738 Paddington Lane	Waipahu
7	1384	VIP Divers Club	32 Main St.	Christianste
8	1510	Ocean Paradise	PO Box 8745	Kailua-Kona
9	1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	Bogota
10	1551	Marmot Divers Club	872 Queen St.	Kitchener
11	1560	The Depth Charge	15243 Underwater Fwy.	Marathon
12	1563	Blue Sports	203 12th Ave. Box 746	Giribaldi

This is DBFBROW in action as it's meant to be.

Figure B



Program Alert

Error: Data type mismatch. Expecting: Character

File: C:\IVDB\9608\5\DBFBROW.PRG
Routine: DBFBROW::SHOW
Line: 30

Ugliness and a less-than-helpful error message result when you change the type of the TableName property.

INSIDE VISUAL dBASE™

Inside Visual dBASE (ISSN 1084-1970) is published monthly by The Cobb Group.

Staff:

Contributing Editor-in-Chief	Keith G. Chuvala
Associate Editors-in-Chief	Jeff E. Davis
	Tiffany M. Taylor
Publications Coordinator	Maureen Spencer
Editors	Laura Merrill
	Joan McKim
	Elisabeth Pehlke
Production Artist	Alison Schwarz
Product Group Manager	Mike Stephens
Circulation Manager	Mike Schroeder
Associate Publisher	Mark Kimbell
VP/Publisher	Mark Crane
President/CEO	J. Thomas Cottingham

Address: Please send tips, special requests, and other correspondence to

The Editor, *Inside Visual dBASE*
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

E-mail address: visual_dbase_win@merlin.cobb.zd.com

For subscriptions, fulfillment questions, and requests for bulk orders, address your letters to

Customer Relations
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

E-mail address: customer_relations@merlin.cobb.zd.com

Phone:

Toll free US	(800) 223-8720
Toll free UK	(0800) 961897
Local	(502) 493-3300
Customer Relations Fax	(502) 491-8050
Editorial Department Fax	(502) 491-3433

Back Issues: To order back issues, call Customer Relations at (800) 223-8720. Back issues cost $8.50 each, $8.95 outside the US. You can pay with MasterCard, VISA, Discover, or American Express, or we can bill you.

Copyright: Copyright © 1996, The Cobb Group. All rights reserved. *Inside Visual dBASE* is an independently produced publication of The Cobb Group. The Cobb Group reserves the right, with respect to submissions, to revise, republish, and authorize its readers to use the tips submitted for personal and commercial use.

The Cobb Group and its logo are registered trademarks of Ziff-Davis Publishing Company. *Inside Visual dBASE* is a trademark of Ziff-Davis Publishing Company. dBASE, dBASE IV, dBASE for Windows, and Visual dBASE are registered trademarks of Borland International. Windows is a registered trademark of Microsoft.

Postmaster: Second Class Postage Paid in Louisville, KY.

Postmaster: Send address changes to:

Inside Visual dBASE
P.O. Box 35160
Louisville, KY 40232

Advertising: For information about advertising in Cobb Group journals, contact Tracee Bell Troutt at (800) 223-8720, ext. 430.

Bulk Sales: For information about bulk/group subscription sales, please contact Customer Relations at (800) 223-8720.

Prices:

Domestic	$79/yr ($8.50 each)
Outside US	$89/yr ($8.95 each)

2041

SECOND CLASS MAIL

Borland Technical Support
Technical Support Information Line: (800) 523-7070
TechFAX System: (800) 822-4269

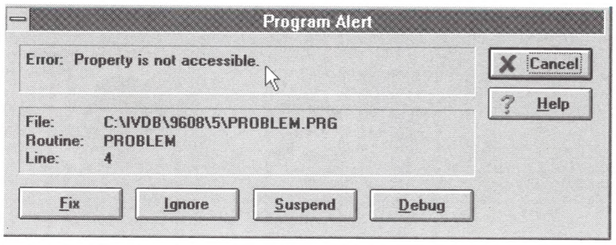
Please include account number from label with any correspondence.

message, as displayed in Figure C. PROTECT doesn't save us from making errors, but it *does* enable dBASE to tell us the location of the real problem in our code. We can now go right to the cause of the error.

PROTECTed properties are also hidden from view in the object inspector, even in the Form Designer. The inspector normally displays unprotected properties on the Properties tab. PROTECTing properties hides them from view. This behavior makes a lot of sense, since the main goal of PROTECT is to prevent unwanted modification of properties from outside the class. If you create a visual custom class for use on your forms, any data that you PROTECT won't be visible in the Form Designer's Inspector window.

PROTECT is a smart new capability; start making it a part of your custom classes whenever a property is intended for use only inside the class. PROTECT can make error messages arising from logic problems more useful, prevent unwanted clutter and confusion in the Inspector window, and help you create a more solid, bullet-proof class design. ❖

Figure C



```
Error:  Property is not accessible.

File:      C:\IVDB\9608\5\PROBLEM.PRG
Routine:   PROBLEM
Line:      4
```

With PROTECT in force, the error message now tells us that the real problem lies in line 4 of PROBLEM.PRG.

Listing B: *DBFBROW.PRG with protected properties*

```
* DBFBROW: Create a safe on-the-fly BROWSE
*
* Usage:  SET PROCDURE TO DBFBROW
*         x = NEW DBFBROW("table_name","Window Title")
*
* CLASS::SetTableName() can be called to set or reset the table name

CLASS DBFBROW
   parameters cTable, cTitle
   this.tablename = iif(type("cTable") = "C",cTable,"")
   this.title     = iif(type("cTitle") = "C",cTitle,"Table View")
   this.browsing  = .f.

   PROTECT tablename,title,browsing

   if .not. isblank(this.tablename)
      this.show()
   endif

   procedure SetTableName
      parameter cTable
      if type("cTable") = "C"
         this.tablename = cTable
      endif
   return

   procedure SetTitle
      parameter cTitle
      if type("cTitle") = "C"
         this.title = cTitle
      endif
   return

   procedure Show
      if .not. isblank(this.tablename)
         if .not. this.browsing
            create session
            use (this.tablename)
         else
            use
         endif
         this.browsing = .t.
         if .not. isblank(tag(1))
            set order to (tag(1))
         endif
         go top
         browse title (this.title)
      endif
   return

ENDCLASS
```

Printed in USA
This journal is printed on recyclable paper.